OpenMPI and parallel Shift-Dictionary with Doublestep calculation

March 26, 2021

1 Problem

Our current algorithms tries to handle the addition of multiple visibilites at once. To achieve this, we group all visibilities together, that have the same shift-index. For every one of these groups we calculate the shift-vector that is representative of all visibilities in this group and. This is simply the sum of all shift-vectors in this group. Threw modelling we have seen, that in most cases we need N groups, since every shift-index appears at least once, in visibility-clusters of relevant size. We usually store the shift-vectors of all groups in a dictionary. I will call this the shift-dictionary. After that we will use the double-step algorithms described in the next section to update the sub-grids of the image-matrix. Because of the way doublestep works, it will be more efficient to calculate all shift-vectors with column-shift first and process, and then handle all row-shiftable shift-vectors.

Using our current implementation of Spift with Apache-Flink, we calculate the shift-dictionary in every parallel task and then every task then updates its sub-grid (Figure 1). We present hear an idea to calculate the shift-dictionary in parallel, and then use a methode we call double-shift to get ride of redundant complex-additions when calculating the sub-grids.

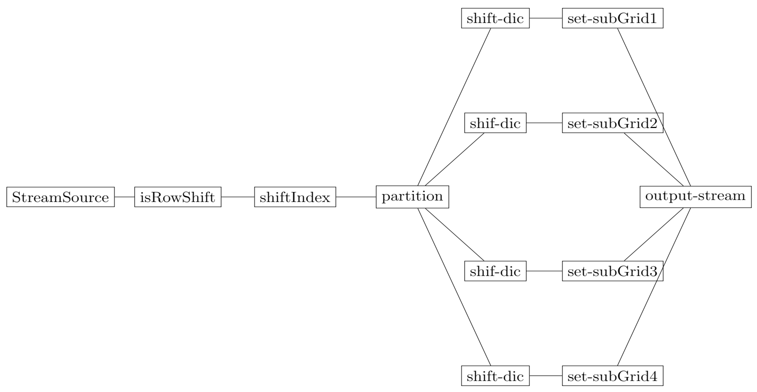

Figure 1: Float-char SPIFT

2 Genreall Idea

We propose to calculate the shift-dictionary only once per machine,and use the parallel doublestep-algorithm we developed using all processes avaliable on one machine. Parallel processes on the same machine could use shared memory for fast access (Figure 2). As long as the processes only share information on the same machine, the communication overhead should be acceptable.

We could not find a way to do this in Apache-Flink. Flink handles the distribution of processes to machines internally and treats all processes the same, independent of where they are running. So communication of nodes on the same machine is not intended and not supported. So we propose to use openMPI and C (or C++). OpenMPI can handle communication between different machines, but it will also allow shared Memory between processes on the same machine.

With this new system, new variables get relevant. The amount of physical machines we call mp, the amount of local processes on machine i we call lp_i and the amount of processes on all machines combined we call gp.

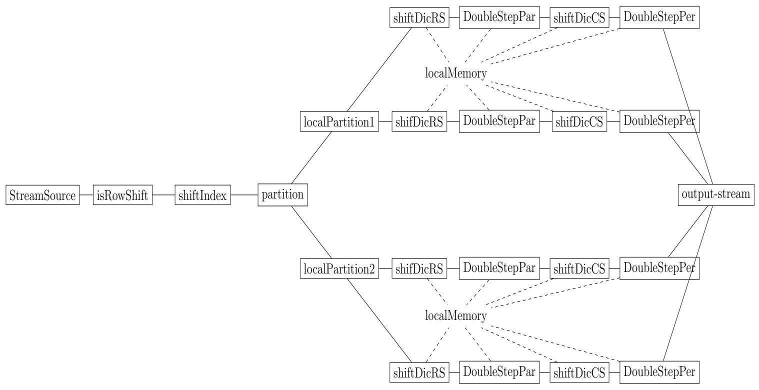

Figure 2: Float-char SPFIT with shared memory and doublestep

3 Doublestep

The naive approach of calculating the final matrix is to add all the shifted values for ervery entry of the matrix. This leads to a lot of additions getting done multiple times. We developed the double-step algorithm to insure every addition only happens once.

3.1 Simple doublestep

Consider two shift-vectors $\vec{v}_1$ and $\vec{v}_2$ with shift-index s_1 and s_2 where $s_1 = s_2 + \frac{N}{2}$ which are both row-shiftable. In the first row, $\vec{v}_1$ and $\vec{v}_2$ get added together $\vec{v}_1[0] + \vec{v}_2[0]$ ([*] denotes by how much the vector is shifted). On the next row, we get $\vec{v}_1[s_1] + \vec{v}_2[s_2]$. On the third row, it's $\vec{v}_1[2 * s_1] + \vec{v}_2[2 * s_2] = \vec{v}_1[2 * s_1] + \vec{v}_2[2 * s_1 + 2 * \frac{N}{2}] = \vec{v}_1[2 * s_1] + \vec{v}_2[2 * s_1] = (\vec{v}_1[0] + \vec{v}_2[0])[2 * s_1]$. And on the forth $\vec{v}_1[4 * s_1] + \vec{v}_2[4 * s_2] = (\vec{v}_1[s_1] + \vec{v}_2[s_1 + \frac{N}{2}])[2 * s_1]$.

As we can see, we only need to make two vector-additions, $\vec{v}_1[0] + \vec{v}_2[0]$ and $\vec{v}_1[s_1] + \vec{v}_2[s_2]$. These two rowse form a $2 \times N$-matrix. If we shift this matrix by a multiple of $2 * s_1$, it corresponds to the shifted addition of v_1 and v_2. We will call this matrix $sm^2_{2*s_1}$, where the subscript denotes the shift-index of the shift-matrix, and the superscript denotes the amount of rows it has. We can do this with all shift-vecor-pairs to get $\frac{N}{2}$ shift-matrices, that all have an even shift-index.

Using all the $2 \times N$-shift-matrices, we can use the same methode to construct $\frac{N}{4}$ $4 \times N$-matrices with shift-vectors that are multiple of four. We repeat this process until we have calculated sm_0^N, which is the final matrix.

3.2 Doublestep with parallelism

As with the naive approach, using a parallel approach means slicing the image matrix into sub-matrices. Different then the naive approach though, all processes on the same machine handle one large slice, rather then one smaller slice per process on the machine. When setting up the pipeline, it still has to be decided if the image-matrix gets horizontally or vertically. Here I will assume it gets sliced horizontally, but the same system will also work with vertical slices.

Calculating the new image matrix slice when a new batch gets processed is split in two halfs. First all shift-vectors with horizontal shift get added, then all shift-vectors with vertical shift. This order is arbitrary. Doublestep works differently, depending if the shift is parrallel to the slicing or perpendicular to it. The different algorithms get discussed in the next two sections.

3.2.1 Parrallel

In this case, we consider row-shift and horizontal slicing. Let $D_i = \frac{N * lp_i}{gp}$ the size of the image-sub-grid that the machine i has to calculate and $l_{i,0} = \sum_{j=0}^{i-1} lp_i + 1$ be the first line that this machine has to calculate (we start counting from 0). In this case we use doubleshift to caculate the first $\log D_i$ steps of doubleshift to get $\log (N - D_i)$ matrices of size $D_i \times N$.

We need all shift-vectors for this, so we have to calculat ethem beforehand. Since it is easiert to calculate the shift-matrices if we assume we are handling the first row of the matrix, we will also shift all values $l_{i,0}$ times, to emulate this. The calculation of the shift-vectors is evenly distributed among all local processes. Every process writes the result its shift-vectors to a block of memory of size $\frac{N^2}{lp_i}$, that only it can write on, but others can read as well.

Algorithm 1 shows how process $k \in \{0, 1, 2, ...(lp_i - 1)\}$ calculates the shift-vectors.

Algorithm 1 Calculating shift-vectors for parallel double-step

```
1: $r = N/lp_i$
2: $range = [k * r, (k+1) * r]$
3: for vis in visibilities, vis.isRS and $vis.shiftIndex \in range$ do
4:     q = emptyVector
5:     for i ← 0 to N do
6:         $q[i] = vis.value * W^{(i*vis.v - l_{i,0}*vis.shiftIndex)\%N}$
7:     memory.setShiftVector(vis.shiftIndex,q)
```

The local processes now calculate the shift-matrices for each step together. We allocate a second blocks of memory of size $\frac{N^2}{lp_i}$ to each process. The first block is to store the result from the previous step (reading block), and the other block to calculate the next step (writing block). All processes have reading-rights to all memory blocks, but only writing rights to their own. Since every step only needs the shift-matrices of the previous one, after every step the previous reading block can become the new writing block.

Initially, there will be enough shift-matrices so that every process can calculate one or multiple full shift-matrices. Later, processes will calculate shift-matrices together, so the value of the shift-matrices will be distributed over multiple memory-blocks. Algorithm 2 shows how processes $k \in \{0, 1, 2, ...(lp_i - 1)\}$ calculates the shift matrices in the j-th step of doublestep ($1 \leq j \leq \log D_i, j \leq \log \frac{N}{lp_i}$) when it can calculate shift-matrices alone.

When there are more processes then shift-matrixes to calculate, multiple processes can start to calculate the same matrix. This can be done, because the calculation of every row in a shift-matrix is independent from all other rows. Processes still store the partial-shift-matrix in there own memory-block, to avoid problems with parallel access. Algorithm 3 shows how processes $k \in \{0, 1, 2, ...(lp_i - 1)\}$ calculates the shift matrices in the j-th step of doublestep ($1 \leq j \leq \log D_i, j > \log \frac{N}{lp_i}$) when it needs to cooperate with other processes.

In the end, all shift-matrices of size $D_i \times N$ need to be added to the previous image-sub-Grid to. Every process can handle it's own sub-Grid,

Algorithm 2 Partial parallel DoubleStep Algorithm for process k for small j

```
1: $r = N/(2^j * lp_i)$
2: $range = [k * r, (k+1) * r]$
3: $shiftIndexes = range * 2^j$
4: for shiftIndex in shiftIndexes do
5:     $m_1 = memory.getPreviousMatrix(\frac{shiftIndex}{2})$
6:     $m_2 = memory.getPreviousMatrix(\frac{shiftIndex}{2} + \frac{N}{2})$
7:     $firstHalf = m_1.shift(0) + m_2.shifted(0)$
8:     $secondHalf = m_1.shifted(\frac{shiftIndex}{2}) + m_2.shifted(\frac{shiftIndex}{2} + \frac{N}{2})$
9:     $memory.setNextMatrix(shiftIndex, concatenate(firstHalf, secondHalf)$
```

Algorithm 3 Partial parallel DoubleStep Algorithm for process k for big j

```
1: $r = \frac{N}{2^j * lp_i}$
2: $number = int(k * r)$
3: $lines = [(k * r - number) * 2^j, ((k+1) * r - number) * 2^j]$
4: $shiftIndex = number * 2^j$
5: for line in lines do
6:     if $line < \frac{2^j}{2}$ then
7:         $l_1 = memory.getLineOfPreviousMatrix(\frac{shiftIndex}{2}, line)$
8:         $l_2 = memory.getLineOfPreviousMatrix(\frac{shiftIndex}{2} + \frac{N}{2}, line)$
9:         $memory.setLineOfNextMatrix(shiftIndex, line, l_1 + l_2)$
10:    else
11:        $l_1 = memory.getLineOfPreviousMatrix(\frac{shiftIndex}{2}, ((line - \frac{N}{2}) - (\frac{shiftIndex}{2}))\%N)$
12:        $l_2 = memory.getLineOfPreviousMatrix(\frac{shiftIndex}{2} + \frac{N}{2}, ((line - \frac{N}{2}) - (\frac{shiftIndex}{2} + \frac{N}{2}))\%N)$
13:        $memory.setLineOfNextMatrix(shiftIndex, line, l_1 + l_2)$
```

and acess the $D_i \times N$ matrices, since every process has read-access. Let $NrLines$ be the number of lines each process has to handle. Algorithm 4 shows how process k calculates its sub-grid.

Algorithm 4 Partial parallel DoubleStep Algorithm for process k, calculating final

```
1: $lines = [0, NrLines]$
2: for line in lines do
3:     $rowToAdd = memory.getLineOfPreviousMatrix(0, k * NrLines + line)$
4:     for $i \leftarrow 1$ to $\frac{N}{D_i}$ do
5:         $rowToAdd$ += $memory.getLineOfPreviousMatrix(i * 2^j, k * NrLines + line)$
6:     $memory.setLineOfSubGrid(line, rowToAdd + memory.getLineOfSubGrid(line))$
```

3.2.2 Perpendicular

In this case we consider column-shift and horizontal slicing. Let $D_i = \frac{N * lp_i}{gp}$ the size of the image-sub-grid that the machine i has to calculate and $l_{i,0} = \sum_{j=0}^{i-1} lp_i + 1$ be the first line that this machine has to calculate (we start counting from 0).

Like in the parallel-case, the processes will now calculate the shift-dictionary and the steps of doublestep together. What differes from the parallel case, is that all steps of doublestep will get calculated, and that not all lines will be considered, since not all lines matter for the final image-sub-grid (Figure 3). Before we start calculating the steps of doublestep, we calculate what lines are needed in each shift-matrix. To do this, we first consider the final $D_i \times N$-Matrix. It will need all the lines in the range $[l_{i,0}, l_{i,0} + D_i]$. To calculate it, we need the lines $[l_{i,0}, l_{i,0} + D_i]$ of the shift-matrix with shift-index 0 and size $N \times \frac{D_i}{2}$, and the lines $[l_{i,0}, l_{i,0} + D_i] \cup [l_{i,0} + \frac{N}{2}, l_{i,0} + D_i + \frac{N}{2}]$. In generall, to calculate the lines of set l for a shift matrix of size $N \times B$ with shift-index s, we need the following lines from the shift-matrices of size $N \times \frac{B}{2}$: From the matrix with shift index $\frac{s}{2}$ we need the lines of set $l \cup l + \frac{s}{2}$ and from the shift-matrix with shift-index $\frac{s+N}{2}$ we need lines of set $l \cup l + \frac{s+N}{2}$. Using this formula we can recursevly calculate all lines we need for each shift-matrix, as seen in Algorithm 5.

Algorithm 5 Calculating needed lines for shift-matrices

```
1: $MatrixSizeDic = dic()$
2: $D = N$
3: $originalLines = \{l_{i,0}, l_{i,0} + 1, ..., l_{i,0} + D_i - 1\}$
4: $originalShiftMatrixDic = dic(0 : originalLines)$
5: $MatrixSizeDic[N] = originalShiftMatrixDic$
6: $D = \frac{D}{2}$
7: while $D \leq 1$ do
8:     $newLinesDic = dic()$
9:     for shiftIndex in MatrixSiteDic[2*D].keys do
10:        $lines = MatrixSiteDic[2 * D][shiftIndex]$
11:        $newLinesDic[\frac{shiftIndex}{2}] = lines \cup lines + \frac{shiftIndex}{2}$
12:        $newLinesDic[\frac{shiftIndex+N}{2}] = lines \cup lines + \frac{shiftIndex+N}{2}$
13:    $MatrixSizeDic[D] = newLinesDic$
14:    $D = \frac{D}{2}$
```

Now that we know what lines are needed, we can start to calculate them. Since the amount of lines one shift-vector needs can vary by a lot, we can't just allocate the vectors to different processes and assume they will all have to do the same ammout of work. Instead, we will create a list of all values that need to be calculated in all vectors, and then divide that list evenly between processes. Then each process will calculate one equal chunk of that list (Algorithm 6).

Now that the shift-vectors are calculated, we can proceed to calculate the shift-matrices. The process is baseically the same. Make a list of all rows that need to be calculated, divide the list evenly among th processes and then calculate them (Algorithm 7).

In the end, all image-sub-Grid need to be updated. Every process can handle it's own sub-Grid, and acess the final shift-matrix, since every process has read-access. Let $NrLines$ be the number of lines each process has to handle (Algorithm 8).

Algorithm 6 Calculating shift-vectors for parallel double-step by process k

```
1: $list = []$
2: for shiftIndex in MatrixSizeDic[1].keys do
3:     for line in MatroxSizeDic[1][shiftIndex] do list.add( (shiftIndex,line) )
4: $chunkSize = roundedUp(\frac{list.size}{lp_i})$
5: for $i \leftarrow k * chunkSize$ to $(k+1) * chunkSize$ do
6:     shiftIndex, line = list[i]
7:     vis = visibilities.get(shiftIndex)
8:     $memory.set(shiftIndex, line, vis.value * W^{line*vis.u})$
```

Algorithm 7 Calculating the j-th step for perpendicular double-step by process k

```
1: $list = []$
2: for $shiftIndexinMatrixSizeDic[2^j].keys$ do
3:     for $lineinMatroxSizeDic[2^j][shiftIndex]$ do list.add( (shiftIndex,line) )
4: $chunkSize = roundedUp(\frac{list.size}{lp_i})$
5: for $i \leftarrow k * chunkSize$ to $(k+1) * chunkSize$ do
6:     shiftIndex, line = list[i]
7:     if $line < \frac{2^j}{2}$ then
8:         $l_1 = memory.getLineOfPreviousMatrix(\frac{shiftIndex}{2}, line)$
9:         $l_2 = memory.getLineOfPreviousMatrix(\frac{shiftIndex}{2} + \frac{N}{2}, line)$
10:        $memory.setLineOfNextMatrix(shiftIndex, line, l_1 + l_2)$
11:    else
12:        $l_1 = memory.getLineOfPreviousMatrix(\frac{shiftIndex}{2}, ((line - \frac{N}{2}) - (\frac{shiftIndex}{2}))\%N)$
13:        $l_2 = memory.getLineOfPreviousMatrix(\frac{shiftIndex}{2} + \frac{N}{2}, ((line - \frac{N}{2}) - (\frac{shiftIndex}{2} + \frac{N}{2}))\%N)$
14:        $memory.setLineOfNextMatrix(shiftIndex, line, l_1 + l_2)$
```

Algorithm 8 Partial perpendicular DoubleStep Algorithm for process k, calculating final

```
1: $lines = [0, NrLines]$
2: **for** line in lines **do**
3:     $rowToAdd = memory.getLineOfPreviousMatrix(0, k * NrLines + line)$
4:     $memory.setLineOfSubGrid(line, rowToAdd + memory.getLineOfSubGrid(line))$
```

4 Calculation

Doublestep with parallel processes and shared memory takes $\log \frac{N}{gp} * \frac{N^2}{max(\{lp_0,...,lp_{gp-1}\})}$ complex additions and less then $\frac{3}{2}N^2$ complex additions.

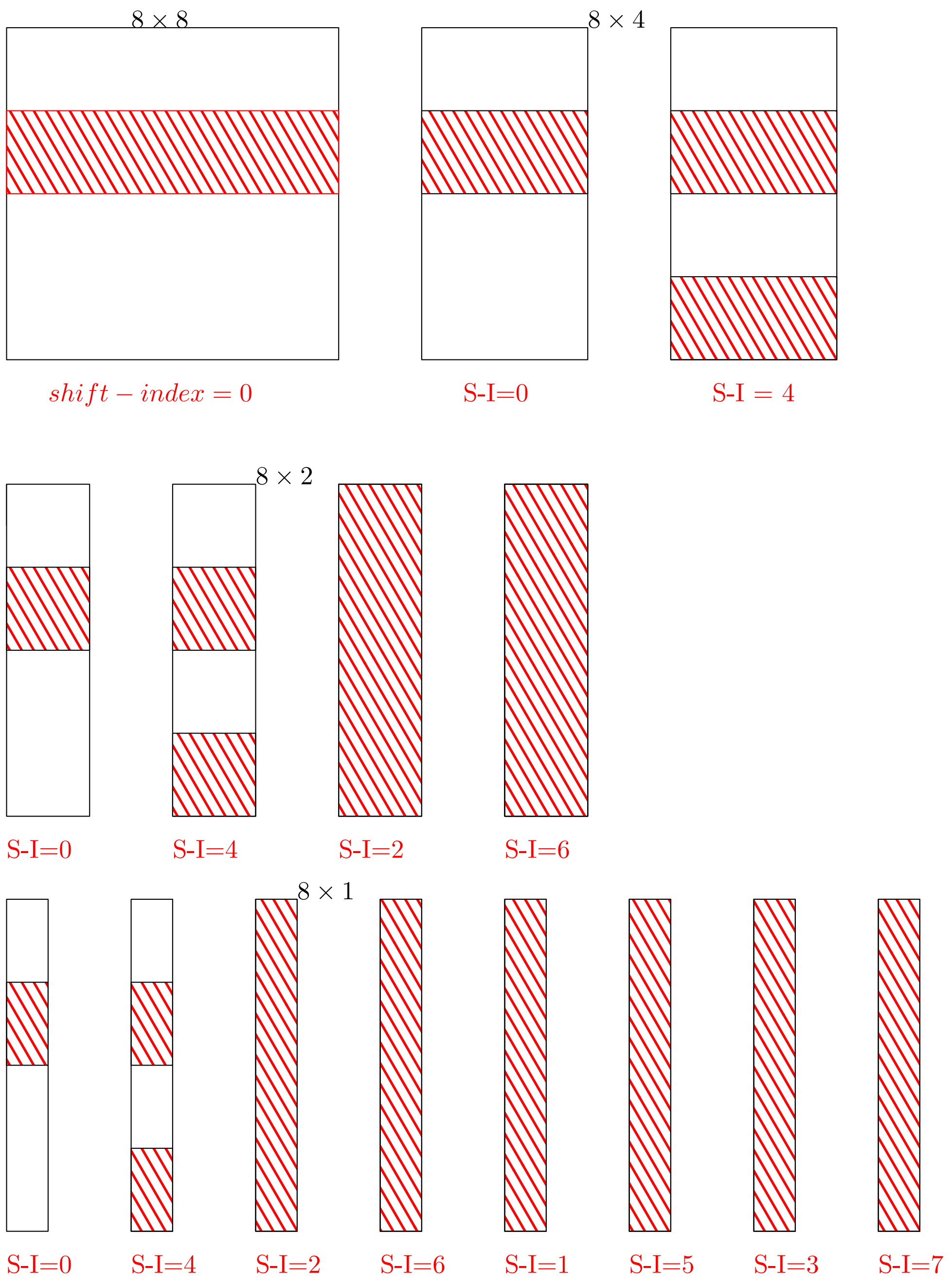

Figure 3: Cool caption